Data Search and Reorganization using FPGAs: Application to Spatial Pointer-based Data Structures

Pedro C. Diniz and Joonseok Park
University of Southern California/Information Sciences Institute
4676 Admiralty Way, Suite 1001
Marina del Rey, California 90292
{pedro, joonseok}@isi.edu

ABSTRACT

FPGAs have appealing features such as customizable internal and external bandwidth and the ability to exploit vast amounts of fine-grain parallelism. In this paper we explore the applicability of these features in using FPGAs as smart memory engines for search and reorganization computations over spatial pointer-based data structures. The experimental results in this paper suggest that reconfigurable logic, when combined with data reorganization, can lead to dramatic performance improvements of up to 20x over traditional computer architectures for pointer-based computations, traditionally not viewed as a good match for reconfigurable technologies.

Keywords
Custom Computing; Data search and Data Reorganization Engines; Hardware support for Pointer Operations; Field-Programmable-Gate-Arrays (FPGAs).

1. INTRODUCTION

The growing gap between processor and memory access speeds means that in order to make good use of current cache-oriented processors, data must exhibit very good locality. Increasing cache sizes and/or set-associativity leads to diminishing returns and incurs significant power penalties. Of particular concern are applications that manipulate pointer-based data structures as they exhibit very irregular data access patterns, rendering caches ineffective. As a result, pointer-based computations typically perform poorly on traditional computer architectures.

A design alternative is to provide smart memory engines based on Field-Programmable-Gate-Arrays (FPGAs) that can scan, select and reorganize large data sets directly in memory. The resulting reorganization, either in-place or using relocation would increase the locality of subsequent computations on the selected data, enhancing the effectiveness of caches and consequently substantially increasing the overall system performance. As FPGAs can implement a wide variety of customizable logic circuits, they can also exploit application specific searches by mapping directly to logic circuits, rather than being implemented as long sequences of instruction, functionality that defines the search criteria. A typical example is the utilization of pattern-matching operations for the selection of sophisticated records in multimedia databases.

In this paper we describe and evaluate the implementation of FPGAs as smart memory engines capable of sophisticated and customizable memory traversal, selection and relocation operations. These engines can be part of an overall computing system as traditional co-processors or can be located directly in memory for increased bandwidth, in a processing-in-memory strategy. In terms of the programming model we propose an approach where the programmer explicitly controls the consistency of the data manipulated by the smart memory engine and the host processor.

This paper makes the following specific contributions:

- It describes the design and evaluation of smart memory engines that support pointer traversals in memory for pointer-based data structures. The designs also support the relocation of data in memory for improved locality.
- It describes the mapping of sophisticated traversals for two important spatial data structures, namely Sparse-mesh and Quad-tree structures, to the proposed data search and reorganization engines.
- It presents experimental results for the implementation of a set of spatial queries over the two spatial data structures on a real FPGA-based system. Our results reflect the utilization of the proposed engines with a traditional host system for single-FPGA and multiple-FPGA scenarios.
- It suggests that reconfigurable logic when combined with data reorganization can lead to dramatic performance improvements over traditional computer architectures for pointer-based computations.

A more general contribution of this paper is that, and despite of their current clock speed handicap, FPGAs can be integrated with traditional architectures to substantially increase the performance of the overall systems for computations that traverse pointer-based data structures. This illustrates the potential for FPGAs for these computations, which have traditionally not being viewed as a good match for reconfigurable technologies.

* Funded by the National Science Foundation (NSF) under Grant CCR-0209228.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

FCCM’03, April 08-11, 2003, Napa, California, USA.
We envision future architectures to be able to integrate modest amounts of reconfigurable logic. A system-level strategy of having the proposed smart memory engines, either inside future processor designs or as detached memory engine co-processors, should allow systems to exhibit better scaling characteristics than caches for application domains that exhibit very irregular data access patterns.

This paper is organized as follows. In the next section we present an example on how to use smart memory engines for the screening and reorganization of nodes of a pointer-based data structure. Section 3 outlines the execution model and system-level issues for using the proposed data reorganization engines in memory and contrast our approach with other intelligent-memory efforts, most notably Active Pages [9]. Section 4 describes a set of important spatial data structures. Section 5 describes the implementation of the smart memory engines that can support traversals on these spatial data structures. In section 6 we present experimental results for the implementation of the proposed smart memory engines with a sample set of spatial pointer-based data structures and spatial queries. We survey related work in section 7 and conclude in section 8.

2. EXAMPLE

We now illustrate the application of the proposed smart memory engines in the screening and reorganization of a computation that traverses a large set of data nodes organized in a multi-way pointer-based tree data structure.

```c
#define NUM_CHILDREN 4

typedef struct record_node { 
  ... /* node data here */ 
  ... struct record_node* link; 
} record_node;

typedef struct tree_node { 
  int kind; /* LEAF = 0, TREE = 1 */ 
  ... /* data of tree/leaf nodes here */ 
  union{
    struct tree_node *child[NUM_CHILDREN]; 
  record_node* record; 
  } rec_field; 
  } tree_node;

void SelectAndLinkTraverse(tree_node *ptr, <args>){ 
  if(isLeafNode(ptr) == TRUE){ 
    if(selectionCriteria(ptr->rec_field.record) == TRUE){
      addToList(result_list,ptr->rec_field.record); 
    }
    } else { /* internal node case */
      if(SkipTree(node)<args>) == FALSE}{ 
      for(i=0; i < NUM_CHILDREN; i++) 
        SelectAndLinkTraverse(ptr->rec_field.child[i], <args>); 
    }
    }
  ... /* go through the selected nodes and perform another computation */
  ptr = result_list; 
  while(ptr != NULL){ 
    doSomething(result_list,ptr); 
    ptr = ptr->link; 
  }
```

Figure 1. Data screening and reorganization example code.

The relevant portions of the code are shown in Figure 1 and consist of a set of data structure definitions for nodes of a multi-way tree. Nodes can be either leaf nodes, as indicated by a zero value of the kind field of the tree_node, or internal nodes otherwise. For leaf nodes, a single pointer field references the data node of the record_node type. Internal nodes can have up to 4 children nodes, either other internal nodes or simple leaf nodes.

The computation is organized as a recursive traversal of the tree along which it gathers a subset of the record nodes at the leaves that meet a given selection criteria. Selected nodes are linked using the link field in their struct definition as implemented by the addToList function in the code. Under some conditions the computation skips an entire section of the tree as dictated by the SkipTree function. In this example we have omitted the details about general traversal arguments <args> for simplicity.

A way to use the configurable logic in the smart memory engines is to place the memory engine as a data screener between memory or even a secondary storage (disk) and the processor. The memory engine will read a large number of data objects and select the ones that match a given criteria. The host processor will only see a very small set of the objects without having to scan them one at a time by traversing linked data structures. In addition the memory engine can relocate the selected objects in memory to substantially enhance the locality, and therefore the performance, of the overall computation on the host.

Figure 2 illustrates this concept. On the left we have the original set of linked nodes. The FPGA scan these nodes and reorganizes the selected ones (via copy) in memory (either in the same or in another bank) on the right hand side. The FPGA acts as a memory engine with custom logic implementing the node selection criteria and supporting the copy functionality. The custom logic also deals with updating pointer fields to reflect new node locations\footnote{While in general changing the location of an object in memory and updating all of the references into it is an infeasible problem, it is possible to engineer a scheme in which the original records, having been relocated can use an additional pointer field to indicate the location of the new object so that subsequent accesses can find its new relocated copy. A similar strategy is used in garbage collection approaches in object-oriented systems.}.
3.1 Execution Model

We view the organization of the smart memory engines as logic that can execute a set of programmable functions over a set of virtual memory pages that are associated with the engine. From the perspective of the host processor each FPGA executes a “hardware thread” with which the host processor synchronizes, either to start or to check for termination using a simple set of program level functions.

Each hardware thread executes a function with a set of arguments passed by value and traverses a pointer-based structure that it does not modify. Rather the computation copies subsets of the data traversed into a selected set of pages. In this model, which we call read-copy-modify (RCM) the data structures the computation traverses must be flushed out of the host processor(s) caches. After the hardware thread terminates the host processor reads the data from the result pages and possibly copies it or some portion of it to other regions. Figure 4 below illustrates this organization.

3. SYSTEM MODEL AND BACKGROUND

We now describe the fundamental execution and programming model of our proposed approach and contrast it with other related proposals such as Active Pages [9] and intelligent-memory research projects.

---

2 Given that FPGAs can have a wide internal memory bandwidth to its memory it is possible to implement very effective implementation of hardware functions to clear the contents of entire pages.
computation completes the operating systems is free to move pages around at will. Pages fault concerns are also avoided using this simple scheme. Finally, host and hardware threads communicate via memory-mapped regions of memory that are non-swappable and non-cacheable.

3.2 Programming Interface
The programming interface is analogous to the interface defined for the Active Pages research project [9], namely functions to allocate and assign memory pages to FPGA devices; functions to define what the function of each FPGA should do; and synchronization functions between the host processor and the hardware threads.

3.3 Other Intelligent Memory Approaches
Although very similar in spirit to the Active Pages research project [9] the work presented here differs in several aspects. First we focus on a model that does not allow the hardware threads to directly modify the contents of the program data. Instead we require the hardware threads to generate its results on scratch memory pages. Our model is that hardware threads should be used to scan-selected-reorganize data and not directly modify it. Active Pages allows direct computation over the data on the pages themselves. As a result of these differences our “error” model is much simplified. The execution can always be restarted or retried by a host processor when a hardware thread terminates abnormally.

Approaches that allow for independent processing units to directly touch memory, such as is the case in Processing-In-Memory (PIM) (e.g., [6][7]), have to contend with abnormal program behavior and cache consistency. They require the host processor to flush the pages that will potentially be modified by the PIMs. This is an additional overhead and not all processors and/or APIs exhibit this functionality.

As with Active Pages we also impose that each hardware thread cannot communicate with other threads (c.f. with the parallel mechanism in the DIVA architecture [6]). The scalability and efficiency of the parallel execution is directly controlled by the granularity and synchronization explicitly defined and controlled by programmers. In approaches that allow direct communication between hardware threads scalability is indirectly defined by the data access patterns and distribution of data structures throughout the memory.

4. SPATIAL DATA STRUCTURES
We now describe two important pointer-based spatial data structures, a Sparse-Mesh and a Quad-Tree. For simplicity we focused on two-dimensional representations although the structures can be easily generalized with the addition of more pointer fields for additional dimensions. We also describe spatial queries over these data structures in C code.

4.1 Sparse-Mesh
The first spatial data structure is a sparse representation of a 2D system of coordinates. Columns and rows represent regions of the 2D space and are structured as equally-spaced bins, i.e., a given column or row has all of the entries with a value of y axis in the range (-y) to (-y+dy) and sorted in ascending values of the x axis coordinate. Similar organization holds for the columns. Some column or row bins are empty corresponding to the absence of any data, either in that column or row. Figure 4 illustrates a Sparse-mesh data structure example and depicts a prototypical spatial query C code given the x and y coordinates of the lower left and upper-right corner of the region of interest.

The sample query uses a simple strategy that scan the entire sparse data structure, first checking if a node is inside the spatial region of interest, and then applying a secondary selection criteria function. If successful, the node is placed in a linked list, possibly using some of the nodes link fields in its type declaration. In this example we have used rectangular spatial regions as defined by the values of the x and y coordinates of a lower-left and an upper-right corners.3

```c
void SpatialQuery(sm *sparse, int x_ll, int y_ll, int x_ur, int y_ur){
  row_node *rn; data_node *dn;
  rn = sm->rows;
  while(rn != NULL){
    dn = rn->first_node;
    while(dn != NULL){
      if(insideRegion(dn,x_ll,y_ll,x_ur,y_ur) == TRUE){
        if(selectionCriteria(dn,x_coor) == TRUE){
          addToList(dn,x_coor);
        }
        dn = dn->next_col;
      }
    }
    m = m->next_row;
  }
}
```

Figure 4. Sparse-Mesh structure and sample query code.

4.2 Quad-Tree
The second spatial data structure organizes the nodes using a pointer-based Quad-Tree data structure [19]. Each of the internal nodes of the Quad-Tree contains all of the nodes, directly as leaf nodes or recursively as internal nodes of any of its descendent nodes that lie in a given quadrant of the space. Each descendent node of an internal node subdivides the region of its ancestor into 4 equal quadrants. Figure 5 illustrates a simple example with 9 nodes. The data structure generalizes to 3D as an Oct-Tree used in physics problem such as the N-body simulation. Figure 5 also presents a sample recursive spatial query over a Quad-Tree data structure. The function SpatialQuery uses the auxiliary function partialOverlap to determine whether or not there are data nodes of interest inside its sub-tree. If not it skips the search on the tree node and returns.

3 It is possible to be smarter about the traversal by determining first in which orientation (columns or rows) should the structure be scanned to minimize the number of accessed rows/columns.
The selection of which data structure to use depends on the distribution of the data in the space of interest and most importantly on the properties of the spatial queries. For spatial queries with squared boundaries, the Quad-Tree should perform well as the internal nodes subdivide the space in squared regions. For general, and more irregular regions, the Sparse-Mesh representation should perform well as it avoids the organizational overhead of the Quad-Tree.

5. SPATIAL QUERIES IN HARDWARE

We now describe hardware structure for supporting the direct execution of spatial queries over the two spatial data structures described in the previous section. Before presenting the specific designs for these data structure we first introduce the general hardware infrastructure as a memory engine with customizable control and with support for pointer chasing. These basic features will later be used to describe the more advanced implementations for the spatial queries of the data structures described previously.

5.1 Basic Memory Engine Architecture

We now describe the internal structure of the proposed smart memory engine depicted in Figure 6. It consists of a memory interface unit that deals with the physical layer of the memory; a controller that deals with scheduling of the memory operations such as basic read and write; conversion FIFOs that allow for very wide internal buses to be converted in a sequence of word memory operations; and a custom logic unit where the custom memory operations are implemented. The custom logic unit has a set of registers that are visible to an outside entity and typically memory mapped onto a host address space.

The design in Figure 6 is an evolution of our previous memory controller designs [11] with the addition of specific support for pointer dereferencing and bounds checking as detailed in the next section.

```
void SpatialQuery(node *tree, int x_ll, int y_ll, int x_ur, int y_ur){
    if(is_leaf_node(tree)){
        if(selectionCriteria(tree->data,<args>)) == TRUE{
            addToList(result_list,tree);
        }
    }
    if(partialOverlap(tree,x_ll, y_ll, x_ur, y_ur)){
        for(int i = 0; i < 4; i++){
            SpatialQuery(tree->ptr[i], x_ll, y_ll, x_ur, y_ur);
        }
    } else { /* skip as there are no relevant nodes in this region */
    }
}

Figure 5. Quad-Tree structure and sample query code.
```

In our design approach we have chosen an overall system architecture where the memory engine can act autonomously much like DMA engines. Other arrangements, such as the possibility of embedding the FPGA-like fabric as part of the processor core exhibit complications. First, the engines will interfere with the internal processor memory interface reducing its external memory bandwidth. Second, an interface to caches needs to be arranged if the smart memory engine is to modify data in memory. Overall any hardware implementation that is not flexible enough to allow the definition of regions of the address space where consistency is not required, will lead to potential performance degradation either in the form of longer implementation critical paths or unneeded cache-memory traffic. The software solution proposed here essentially corresponds to a cache consistency model in which address regions are explicitly managed by the compiler or in its absence the programmer.

5.2 Pointer Chasing and Data Reallocation

As with traditional processor architectures, scanning irregular pointer-based data structure in memory requires the ability to dereference virtual address stored in memory. In this discussion, and given the focus of this work on relocation of data to and from a virtual address space to another address space, we call the original address space where the data is originally located the "virtual" address space, and the address space where the copy of the data or the relocated data is placed the "local" address space.

Given a pointer-field in memory dereferencing its value amounts to simply reading its contents and addressing the memory with its value as an address. In the context of the relocation operation the implementation must take into account the fact that a block of data was relocated from an original virtual address to another address space based on a given "local" base address.

Figure 7 depicts this relocation mapping and presents a hardware solution to effectively find a new "local" address corresponding to a virtual address extracted from relocated data. On the left-hand-side we have a generic data structure with a set of data fields and two pointer fields. To determine the "local" address of the data pointed to by a given field the implementation must compute the relative offset from relocation. This relative offset is calculated by first subtracting the pointer contents against a virtual base address and then adding the result to the local base offset value. The subtraction step is used to determine the relative
virtual address of the data inside the allocation unit that was relocated from the virtual address space to the local address space. Along this translation process the implementation must check the validity of a pointer address, by first determining if it is a null value and then if it points to an invalid address region outside the range of local addresses.

Several compiler level transformations can be implemented to improve the performance of this implementation as carried out in our experimental performance results. First, the compiler can reorder the fields in each record so that the selected link link pointer field is presented as soon as possible during the records reading phase, to perform address translation without delaying the computation that determines the selection of the record for copying. Second, for the string pattern matching can be improved by not only allowing the various patterns to be tested concurrently, but also by using a divide-and-conquer algorithmic technique that effectively cuts a linear comparison with N cycles to N/2 + (M-1) where N is the size of the longer string and M the size of the pattern to be tested.

The particular choice of which of these technique to apply depends on the size of the record being read, and therefore the amount of clock cycles it takes to load a full record, the size of the patterns and the amount of hardware to be used in the particular string matching computation.

5.4 Design for Quad-Tree Queries

For the implementation of the queries using the Quad-Tree data structure organization, we have used a limited stack to explicitly hold the relevant traversal data in each of the nodes during the recursive selection along the Quad-Tree. This strategy trades off space on the FPGA by saving the data in each node, rather than its physical address with the time it would take to fetch the corresponding node from memory in order to proceed with the traversal. In the current experiments we have chosen to save in the stack the only the relevant internal node data for the purposes of the traversal. In the future, with FPGAs with denser internal RAM blocks, regular registers space will become less of an issue.

Figure 9 depicts the organization of the hardware implementation for the selection and relocation of nodes in a Quad-Tree data structure. Because there are multiple pointer links to be potentially followed, the implementation cannot easily exploit the overlapping of computation with memory operations. Although speculative techniques could be used we have not explored them at this point.

In this implementation, as in the previous Sparse-Mesh organization, we have used a parallel version of the string matching computation to accelerate the evaluation of the selection of each of the accessed records through the Quad-Tree walk, whenever applicable.
6. EXPERIMENTAL RESULTS

We have implemented code generation functions in C that emit templates in structural and behavioral VHDL for the complete memory engines with custom logic as described in section 4. In this section we present experimental results of the application of the data search and reorganization engines to the spatial data structures described in section 3. We compare the performance of single-FPGA and multi-FPGA engines against a software-only implementation of the same computation on a commercially available desktop system.

6.1 Application Experience

For our application experience we have used a computation inspired on a real life computational biology application. In this application DNA sequences are arranged in a 2D coordinate system resulting from the extraction of the sequences using a micro-array [14]. We now describe the data sets and queries used to evaluate the performance of our implementations. For this experience we have developed a C application that is about 1,300 lines of source code and implemented all of the spatial queries described below.

6.1.1 Data Representation and Data Sets

The data corresponding to an individual DNA element is organized as a data record. Each record is identified by a single id number and has a creation date and update date along with the lower-left corner and upper-right corner coordinates where the strand of DNA is located. In this experiment we limited the size of the DNA sequence to 80 characters. Each record has, therefore a total size of 152 bytes.

We used several data set sizes, ranging from the small data set sizes with 1024 (1K), DNA sequences, to large data set sizes with 262,144 (256K) DNA sequences distributed in a squared grid of 16384 by 16384. Figure 10 depicts the distribution for the 2K data set. For the other data set sizes samples we observe similar spatial distributions.

![Figure 10. Spatial distribution of records and selection region for 2K data set.](image)

To evaluate the impact the reorganization in memory for these data structure would have, in an application executing one of these queries, we have augmented the original application code with a sorting computation over the selected records. The sorting code creates an auxiliary sorting index array (rather than copying the records in place) based on the numerical value of one of the fields of the records and uses the Bubblesort algorithm.

6.2 Desktop Performance Results

We begin this discussion with the wall-clock time results for executing the queries for the two spatial data structures on a SUN Blade 100 workstation. This target desktop system is fitted with a 64-bit UltraSparc Ile processor clocked at 500 MHz using splitied 16Kbyte L1 I-cache, a 16Kbytes D-cache (2-way set associative), a unified 256KByte 4-way associative L2 cache and with a 1GByte of RAM. The results in table 2 are the average of the 3 median values of 5 trial runs where we eliminated the best-case and worst-case values.

Table 1. Number of selected records per query/data set.

<table>
<thead>
<tr>
<th>Data set</th>
<th>Size (Mbytes)</th>
<th>Number of selected records</th>
</tr>
</thead>
<tbody>
<tr>
<td>1K</td>
<td>0.15</td>
<td>16 (1.5%)</td>
</tr>
<tr>
<td>2K</td>
<td>0.30</td>
<td>28 (1.4%)</td>
</tr>
<tr>
<td>4K</td>
<td>0.61</td>
<td>47 (1.1%)</td>
</tr>
<tr>
<td>8K</td>
<td>1.22</td>
<td>85 (1.0%)</td>
</tr>
<tr>
<td>16K</td>
<td>2.43</td>
<td>186(1.1%)</td>
</tr>
<tr>
<td>32K</td>
<td>4.86</td>
<td>376 (1.1%)</td>
</tr>
<tr>
<td>64K</td>
<td>9.73</td>
<td>725 (1.1%)</td>
</tr>
<tr>
<td>128K</td>
<td>19.46</td>
<td>1,506 (1.1%)</td>
</tr>
<tr>
<td>256K</td>
<td>38.91</td>
<td>3,031 (1.2%)</td>
</tr>
</tbody>
</table>

Table 2. Wall-clock time on a SUN Blade 100 (msecs)

<table>
<thead>
<tr>
<th>Data Set</th>
<th>Sparse-Mesh</th>
<th>Quad-Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query</td>
<td>Sorting</td>
<td>Query</td>
</tr>
<tr>
<td>1K</td>
<td>0.92</td>
<td>0.008</td>
</tr>
<tr>
<td>2K</td>
<td>1.77</td>
<td>0.019</td>
</tr>
<tr>
<td>4K</td>
<td>3.60</td>
<td>0.054</td>
</tr>
<tr>
<td>8K</td>
<td>7.14</td>
<td>0.195</td>
</tr>
<tr>
<td>16K</td>
<td>14.2</td>
<td>1.88</td>
</tr>
<tr>
<td>32K</td>
<td>29.12</td>
<td>9.99</td>
</tr>
<tr>
<td>64K</td>
<td>59.90</td>
<td>42.63</td>
</tr>
<tr>
<td>128K</td>
<td>122.9</td>
<td>253.2</td>
</tr>
<tr>
<td>256K</td>
<td>254.4</td>
<td>898.7</td>
</tr>
</tbody>
</table>

Table 2 reveals that, surprisingly, the Sparse-Mesh outperforms the Quad-Tree implementation. We have investigated this apparent “abnormal” behavior and observed that for spatial regions that match better the Quad-tree representation, the Quad-Tree outperforms by far the Sparse-mesh. We concluded, therefore, that the spatial region used in the sample query elicits poor Quad-Tree performance and is likely to be due to the fact that it covers non-trivial regions of the Quad-Tree quadrant organization.
To further understand the behavior of these queries we evaluated the cache characteristics, using the Shade instruction-tracing tool for a system with the same cache parameters as the target SUN workstation system. The results of this study show that the L1 cache miss rate is approximately 48% for the Sparse-mesh and 32% for the Quad-tree. As for the L2 cache the miss rates are respectively 66% and 96%. This confirms the extremely poor cache performance of this application for both data structures.

### 6.3 FPGA Implementation Results

We report results for two scenarios, respectively a single-FPGA and a multi-FPGA system. The single-FPGA system is intended to observe the raw performance of a single FPGA device with today’s technology against a single host system. The multi-FPGA scenario aims at comparing the performance of a multi-FPGA system against a single host but maintaining a comparable number of transistors assigned to computation in both systems. Given the reported transistor counts for a Ultraspice Ile processor of 23 million and approximately 1.2 million equivalent transistor gates for a Xilinx Virtex® XCV1000BG560 FPGA part we use a maximum of 32 FPGAs in our multi-FPGA scenario.

#### 6.3.1 Reference FPGA-based System

We used the ISE tool set [18] and the commercially available synthesis tools Synplify [14] to generate a real implementation on a Virtex® FPGA device. We then tested the implementations of the computations selecting records for the various data organizations and realloting them in memory on an existing real FPGA-based board between two external memories of the same FPGA device on the WildStar™ board [16].

Table 3 below presents implementation metrics on a single Virtex® FPGA device for the spatial queries for the two data structure organizations described earlier.

<table>
<thead>
<tr>
<th>Table 3. Implementation of spatial queries for Sparse-Mesh and Quad-Tree data structures on a Virtex® FPGA.</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPGA area (100% = 12,288 slices)</td>
</tr>
<tr>
<td>---------------------------------</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Sparse-Mesh</td>
</tr>
<tr>
<td>Quad-Tree</td>
</tr>
</tbody>
</table>

The designs occupy a large fraction of the FPGA (above 75%) where the dominant component is the Selection Logic. This component includes the wide internal registers to accommodate the long data nodes of the various data structures. For the Quad-Tree implementation a 32-node deep stack consumes only about 6% of the overall device and less than 10% of the design. Both designs exhibit respectable clock rates for generator-based designs that use that do not exploit the target FPGA architecture features.

We have used the extracted simulation cycle counts and used the synthesized design for determining the exact wall clock time execution of the various queries. We validated a few of the single-FPGA and multi-FPGA designs on the WildStar™ board for some of the small data set sizes.

#### 6.3.2 Single-FPGA Implementation

For the Sparse-Mesh data structure the implementation is completely memory bound despite an aggressive pipelining execution strategy. Loading each 152-byte long node from memory takes a total of 51 clock cycles, 38 clock cycles in pipelined memory access node, and the remainder 12 to deal with pointer addressing and generic memory signaling. We also took advantage of the performance benefit of overlapping computation of a given node with the prefetching of the subsequent node again in a pipelined fashion. Nevertheless, and given the substantial computation per node the query requires, the overall execution time is still dominated by the dereferencing of pointer links and reading data from memory.

The implementation for the Quad-Tree data structure search is not amenable to be pipelined as following the links on each internal node depends on the computation on that node. As such the implementation needs to fetch the entire node resulting in a design that is essentially memory bound.

The results in table 4 reveal that, despite the comparatively low clock rates, the query execution on a single FPGA is on par with the software-only implementation on the SUN workstation (see table 2). We attribute this fact to the application’s poor cache performance as revealed by our cache simulation results (see section 6.2).

#### 6.3.3 Multi-FPGA Implementation Strategy

In this scenario we have assumed a distribution of each of the data structures as follows. For the Sparse-Mesh we distributed the rows of the mesh throughout the memories associated with the various FPGA engines in a blocked fashion. For the Quad-Tree we opted for a distribution in which an entire sub-tree would fit in the local memory of an FPGA-engines. While this leads to the fact that all of the pointer-chasing is localized to that sub-tree and hence can be easily parallelized it exhibits the potential problem of load imbalance. This imbalance is due to the fact that some of the spatial queries might lead to FPGAs not having enough records in the region of interest and therefore being idle. The alternative of doing a very fine-grain distribution by selection the smaller sub-tree closer to the leaves of the whole tree, would balance better any random query but would require inter-FPGA communication which our model does not support. We have investigated this lack of scalability and found out that indeed the strategy for parallelization described above suffers from workload imbalance problems.

#### 6.3.4 Performance Comparison

In the single FPGA scenario, the implementation of the query on the FPGA trails the SUN system by a 50% to 100% gap, i.e. a slowdown of a factor of 2 at most. When combined with the reorganization, however, the performance of the overall computation using a single FPGA is almost on par for the small data sets and almost 2x better for the large data sets. In fact for the larger data set sizes the cache locality benefits from the reorganization outweigh the slight performance loss in the computation of the query. This is not surprising given the extremely low cache effectiveness for these queries, most noticeable for the larger data sets.

The multi-FPGA implementations reveal a substantial performance improvement of 18x for the Sparse-Mesh and 12x for the Quad-Tree over the SUN implementations. These performance speedups result from both gains in the query...
itself through simple coarse-grain parallelism and from data reorganization. For the Sparse-Mesh the speedup is sustained for exponentially increasing data set sizes revealing constant parallel efficiency. For the Quad-Tree the speedups are lower but reveal similar efficiency behavior. We investigated the reasons for the lower speedups for the Quad-Tree case and have concluded that it is due to load balancing issues. Due to our data partitioning strategy for the large data sets the asymmetry between completion-time between the various FPGAs increases (data omitted for space constraints).

Table 4. Execution time for query and sorting for SUN Blade 100 system and FPGA-based engines for the Sparse-Mesh and Quad-Tree data structures (miliseconds).

<table>
<thead>
<tr>
<th>Data Sets</th>
<th>SUN</th>
<th>SUN w/ FPGA-based Engines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query</td>
<td>Sorting</td>
<td>Query Single FPGA</td>
</tr>
<tr>
<td>1K</td>
<td>0.92</td>
<td>0.008</td>
</tr>
<tr>
<td>2K</td>
<td>1.77</td>
<td>0.019</td>
</tr>
<tr>
<td>4K</td>
<td>3.60</td>
<td>0.054</td>
</tr>
<tr>
<td>8K</td>
<td>7.14</td>
<td>0.195</td>
</tr>
<tr>
<td>16K</td>
<td>14.2</td>
<td>1.88</td>
</tr>
<tr>
<td>32K</td>
<td>29.12</td>
<td>9.99</td>
</tr>
<tr>
<td>64K</td>
<td>59.90</td>
<td>42.63</td>
</tr>
<tr>
<td>128K</td>
<td>122.9</td>
<td>253.2</td>
</tr>
<tr>
<td>256K</td>
<td>254.4</td>
<td>898.7</td>
</tr>
</tbody>
</table>

Figure 11. Speedups for single-FPGA and 32-FPGAs.

In order to keep some of the comparisons fair we have limited the memory bandwidth of the FPGA designs to 32 bits and with a very conservative clock rate of 66MHz. Given that the bandwidth of an FPGA to an internal memory for an FPGA-based smart engines is extremely high and way beyond what the current processor packages would allow, the approach suggested here can achieve even higher speedups.

7. RELATED WORK

We now describe related work (not addressed in section 3.3) in data reorganization and relocation using FPGA-based computing engines as well as the implementation of database queries using FPGAs.

7.1 Data Mapping for FCCMs

Other researchers have addressed the issues of mapping variables to memories and improving the performance of the memory subsystem for custom computing machines. Gokhale and Stone [5] proposed an automatic compile-time array allocation algorithm for multi-level memory subsystem. They attempt to allocate array variables to memories based on the memory latency, data access frequency and execution schedule. In the area of embedded systems researchers have also developed methodologies to automate the mapping and subsequent management of data across multiple memory modules. Catthoor, Balasa et. al. developed and evaluated memory optimizations for embedded systems for particular applications [1,3] and focused on optimizations of memory area and power.

7.2 Data Reorganization

Other researchers have recognized the value of data reorganization for general purpose computing. Rivera and Tseng [12] have implemented a data reorganization compiler analysis and code generation to improve cache locality and memory bandwidth utilization through a combination of packing and padding of multidimensional arrays. In the embedded high-performance computing community there has been a substantial effort in providing a standard data reorganization API [2] with emphasis on programmer and application portability.
7.3 Pointer Support & Address Generators

Data access modules and module generators have been extensively used as a way to manage the complexity of mapping computations to FPGAs. Miranda et al. [8] proposed a method to optimize address stream generations for a single computing task. In our own previous work [11] we developed a set of parameterizable address generation units capable of handling affine data streams and exploiting pipelined memory access features over a set of data channels.

Other researchers have focused on providing, generic pointer support for heap allocated data structures. Séméria et al. have shown how to map a series of C programming language constructs that manipulate pointer variables and combine them with special-purpose heap memory allocators to facilitate the mapping of heap structure to registers [13].

7.4 Database Operations and FPGAs

Other researchers have also recognized the value of having FPGAs to accelerate the implementation of relational database operations. In [17] the authors sketch an example of an application using a mix of image and text elements but do not present an overall implementation for the translation of SQL queries to FPGAs. Their experimental evaluation methodology is limited as they neither address the structure of their hardware implementation nor present a performance analysis of the queries in traditional architectures.

7.5 Discussion

Our work differs from previous efforts in two fundamental aspects. First we focused on selected spatial pointer-based data structures rather then on the development of generic hardware structures for pointer constructs. Second, we have implemented smart memory engines focus and data selection and reallocation rather than on computation. Our programming model supports read-and-copy operations rather than generic in-memory computations. This model, though restricted, offers a simple error-recovery approach and is free from cache-consistency issues. The overall system architecture of the proposed memory engines makes them suitable to be integrated in traditional systems and very likely as solutions in future SoC systems [4].

REFERENCES


